ISSN 2277-2685
IJESR/July. 2022/ Vol-12/Issue-3/1-7
T HARI BABU *et. al.,* / International Journal of Engineering & Science Research

The Case of the Operating Mode of the Skipper-II Parallel Programming Environment for Handling Algorithmic Skeleton Nesting Requirements in Real-World Image Processing Applications

T HARI BABU[1] , Dr. S.M. LAKSHMI SRI[2]
Assistant professor[1,2]
DEPARTMENT OF ELECTRICAL AND ELECTRONICS ENGINEERING
P.B.R.VISVODAYA INSTITUTE OF TECHNOLOGY & SCIENCE
S.P.S.R NELLORE DIST, A.P , INDIA , KAVALI-524201

ABSTRACT

Skipper is a Skeleton-based Parallel Programming Environment being developed since 1996 and running at LASMEA Lobooratory, the Blaise-Pascal University, France. The main goal of the project was to demonstrate the applicability of skeleton-based parallel programming techniques to the fast prototyping of reactive vision applications. This paper deals with the special features embedded in the latest version of the project: algorithmic skeleton nesting capabilities and a fully dynamic operating model. Throughout the case study of a complete and realistic image processing application, in which we have pointed out the requirement for skeleton nesting, we are presenting the operating model of this feature. The work described here is one of the few reported experiments showing the application of skeleton nesting facilities for the parallelisation of a realistic application, especially in the area of image processing. The image processing application we have chosen is a 3D face-tracking algorithm from appearance.

Keywords :

parallel programming, image processing, algorithmic skeleton, nesting, 3D face tracking.

INTRODUCTION

At Laboratories des Sciences et Materiaux pour l'Electroni- ´ que, et d'Automatique (LASMEA), the Blaise-Pascal University's laboratory of electrical engineering, France, we have been developing since 1996 a parallel programming environment, called Skipper (Skeleton-based Parallel Programming EnviRonment), based on the use of algorithmic skeletons to provide application programmers with a mostly automatic procedure for designing and implementing parallel applications. The Skipper project was originally envisoned to build realistic vision applications for embedded platforms. Due to the features in the latest developed version of Skipper, called Skipper-II, it has now turned into a more usable parallel programming environment addressing PC cluster architectures and different kinds of applications as well. The reason to develop such an environment is that, relying on parallel machines, programmers are facing several difficulties. Indeed, in the absence of high-level parallel programming models and environments, they have to explicitly take into account every aspect of parallelism such as task partitioning and mapping, data distribution, communication scheduling, or load balancing. Having to deal with these low-level details results in long, tedious, and error-prone development cycles,1 thus hindering a true experimental approach. Parallel programming at a low level of abstraction also limits code reusability and portability. Our environment finally tries to

"capture" the expertise gained by programmers when implementing vision applications using low-level parallel constructs, in order to make it readily available to algorithmists and image processing specialists. That is the reason why Skipper takes into account low-level implementation details such as task partitioning and mapping, communication scheduling, or load balancing

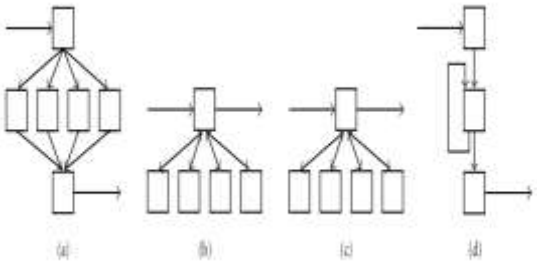

Figure 1: The four skeletons of Skipper are, from left to right, split-compute-merge skeleton, data farming skeleton, task farming skeleton, and iteration with memory skeleton.

The initial implementation of this paradigm was the Skipper-I toolkit (detailed in [1, 2, 3, 4]). However, the variety of skeletons it could accommodate was limited. Irregular skeleton nesting, in which one skeleton might accept another as an argument, was one feature it lacked. To address this issue, the Skipper-II implementation [5] was created. The capacity to manage arbitrarily nested skeletons is its key novel feature. Only a few of projects have developed workable implementations allowing skeleton layering (for examples, see [6, 7, 8]), and skeleton implementers have historically seen this as a challenge. However, the majority of these situations included "toy" applications where skeleton layering is a necessary but "artificial" benchmarking construct. On the other hand, we believe that proving the notion by demonstrating a practical application that needs such a feature in order to be parallelized is of paramount significance.

TheSkipper WORKSHOP

Parallel programming with skeletons and Skipper-I

Methodologies for writing parallel programs based on skeleton code (for examples, see [10, 11]) allow developers to strike a balance between rapid iteration and high performance. Their end goal is to make creating and deploying paral lel apps easier by giving users with direction and an automated process. To that end, they provide a collection of algorithmic skeletons, which are higher-order program structures that encapsulate frequent and frequent kinds of parallelism, making them accessible to the application programmer. Task mapping, data dissemination, communication scheduling, and load balancing are all examples of low-level implementation features that are ignored by the latter. The collection of application-specific sequential functions required to create the skeletons, and the target architecture, are all provided by the application programmer. The total effect is a much shorter period of time between design and implementation and validation. Our major focus is on image processing; hence we've developed and deployed a skeleton-based parallel programming environment (Skipper) based on a collection of skeletons tailored to the needs of vision-centric programs running in parallel [1, 2, 3, 4, 12]. In order to create this collection of starters, we looked backwards at previously written parallel programs. (I) a split-compute-merge (SCM) skeleton; (ii) a data farming (DF) skeleton; (iii) a task farming (TF) skeleton (a recursive variant of the DF skeleton); and (iv) an iteration with memory (ITERMEM) skeleton (as illustrated in Figure 1).

Regular "geometric" processing of iconic data is the focus of the SCM skeleton; in this method, the input set of data is divided into a fixed number of subsets, each of which is processed independently before the final result is obtained by merging the results computed on subsets of the input data (which may overlap). When the number of subgroups is known in advance and the quantity of work to be done on each subset is known in advance, resulting in an equal workload, this skeleton may be used. Convolutions, median filtering, and constructing histograms are all instances of common operations.

The Three-Dimensional Face Tracking Algorithm

Our target application is a viewer-based representation of 3D human faces in picture sequences that can track individual faces. A previously established technique enables for the monitoring of a 2D visual pattern's motion throughout a film. It's fundamental to how we go about things. This approach is explained in detail and tested experimentally in [9]. Face recognition during online meetings is a cool feature.

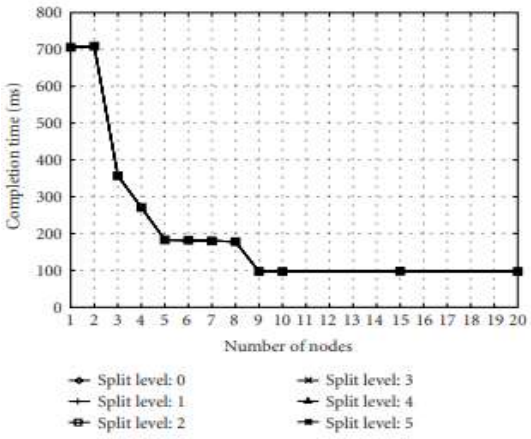

Figure 2: Completion time for the divide-and-conquer benchmark (extract of [16]) (picture size: 512 × 512/8 bits, homogeneous computing power).

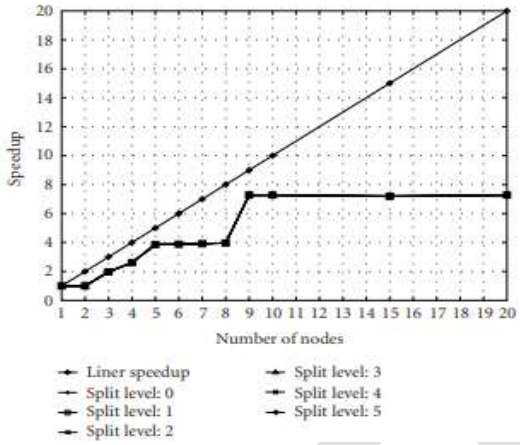

Figure 3: Speedup for the divide-and-conquer benchmark (extract of [16]) (picture size: 512 × 512/8 bits, homogeneous computing power).

In our 3D tracking approach, a face is represented by a collection of 2D images called reference views (appearances to be tracked). Moreover, a pattern is a region of the image defined in an area of interest and its sampling gives a graylevel vector. The tracking technique involves two stages. An off-line learning stage is devoted to the computation of an interaction matrix A for each of these views. This matrix relates the gray-level difference between the tracked reference pattern and the current pattern sampled in the area of interest to its "fronto-parallel" movement. By definition, a "frontoparallel" movement is a movement of the face in a plan

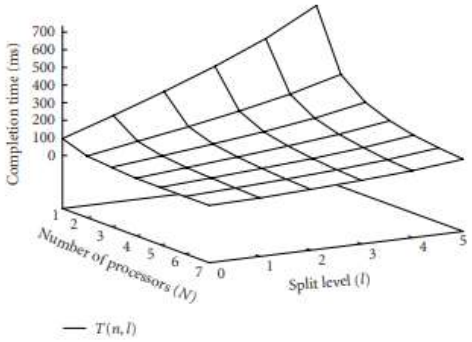

Figure 4: Completion time for the divide-and-conquer benchmark (SKiPPER-I) (extract of [2]). which is parallel to the image's plane.

This shift does not affect the big picture of the pattern standing in for the monitored face. But the placement, direction, and scale of the pattern are all flexible. The on-line stage (Figure 4) involves independently estimating the correlation of ellipse parameters (the target region is supposed to be included in an ellipse) for the current reference pattern and the nearest reference patterns in the collection of images (the previous and the next reference patterns for a movement in roll). Each of these adjustments is achieved by multiplying the gray-level difference between the predicted ellipse's visual pattern and the various reference patterns by the corresponding interaction matrix. We acquire a new location of the ellipse in the current picture that is meant to overlap the actual position of the face for each of these reference patterns that have been examined. The Gray-level difference V I between the current pattern inside the region of interest V Ic and the associated refer ence pattern V Iref is estimated as a quadratic function of location. When switching reference patterns, the one with the least quadratic error shall be used from now on. The ability to switch reference patterns mid-tracking and handle appearance differences owing to roll of the face in the picture are also benefits of this method. We may forego using a face tracking pre-prediction technique since the treatment frequency is more relevant than the face tracking speed. This is supported by the fact that alterations in the pattern's position between two consecutive photos are consistent with those seen during the learning phase.

Implementing Algorithmic Skeleton Nesting for Parallel 3D Face Tracking

Principle

In this context, we are just interested in parallelizing the second step of the tracking algorithm. This may be compared to the following examples. Reference patterns are discussed in further depth in Section 3.2. (1) Previous, current, and future reference patterns may be computed individually and in simultaneously. These calculations are not reliant upon one another and involve the same amount of labour. Therefore, the first level of data parallelism corresponds to the first skeleton shown in Figure 22 (skeleton A). This framework will be used to conduct the parallel comparison of all reference patterns. (2) Additional parallelism in the matrix multiplication stage of processing each reference pattern is possible. This second level of parallelization corresponds to yet another layered data-parallel skeleton (note that the three inner matrix multiplications cannot be collapsed into a single matrix multiplication, therefore the two levels of parallelization cannot be combined). Specifically, one must take into account three distinct interaction matrices (denoted A in the preceding section) and three distinct gray-level vectors (denoted V diff). Only one matrix multiplication of a vector per reference pattern is required. It would be inefficient and lead to inaccurate results to combine the three matrices into a single one, and the same goes for the three vectors. On both scales, data-parallel computations are highly regular (i.e., their complexity is independent of the values of the data they process). This makes them ideal for a system that uses an SCM framework. In this case, the application's parallel structure is composed of two SCM skeletons nested inside one another, with the inner SCM skeleton serving as the compute function for the outer skeleton.

Once the application's parallel structure has been determined, a parallel implementation may be obtained using the Skipper-II environment. Two things are required from the programmer's perspective: (1) defining the parallel structure in a description language (i.e., listing which skeletons are utilized and in what sequence), and (2) giving the application-specific sequential functions to be used as inputs to the skeletons. Earlier Skipper releases used a subset of the Caml language [4] to represent the parallel structure. For Skipper-II, the same strategy is planned. An adapted version of the Camlflow program [19] will be used to construct the intermediate description of the application (in the form of a tree of TF/II skeletons). The current version of Skipper-II still requires human intervention at this stage, since the programmer must provide the intermediate description as a C descriptor for the kernel to utilize directly. This descriptor stores the application's skeleton structure in the form of a C array that corresponds to the tree of TF/II skeletons. Algorithm 4 provides the corresponding descriptor5 for the tracker application (Figure 23 recapitulates the app's skeleton architecture). Each skeleton occupies a single line. A skeleton's "continuation," or whether its results must be sent to an upper level or to another skeleton at the same level,6 is indicated in the second column of each line, and the third column indicates whether this skeleton acts as a slave (i.e., is nested) or as a master.

CONCLUSIONS AND RESULTS

The Intel Celeron Beowulf system (32 x 533 MHz nodes, 100 Mbps switched Ethernet network) was used to run the test. Two different numbers of sampled points in the elliptic region (170 and 373) were used to calculate the completion time of the method and the relative speedup gained in increasing the number of nodes (Figures 24 and 25, respectively). It's important to note that increasing the sample size beyond 170 points does not improve tracking performance (even with only 170 points, tracking is quite accurate). Since this value directly affects the computation/communication ratio, it was raised so that the performance of the Skipper-II kernel could be evaluated further. The Skipper-II ker nel's behaviour may be matched to the three phases of the curves shown in Figures 24 and 25. Those with 1-2 nodes (373 sampled points) or 1-3 nodes (170 sampled points) experience the first phase.

```
S1(
    int          pattern_number,                 /*I   */
    Ellipse      current_ellipse,                /*I/O */
    int  * *     tracker_number_in_test          /*  O */
);
S2(
    Ellipse      current_ellipse,                /*I/O */
    int          tracker_number_in_test,         /*I/O */
    int      *   gray_level_vector_size,         /*  O */
    float    * * gray_level_difference_vector,  /*  O */
    int      * * matrix_line_number,             /*  O */
    float  * * * matrix                          /*  O */
);
P2(
    Ellipse      current_ellipse,                /*I/O */
    int          tracker_number_in_test,         /*I/O */
    int          gray_level_vector_size,         /*I   */
    float    *   gray_level_difference_vector,   /*I   */
    int          matrix_line_number,             /*I/O */
    float    *   matrix,                         /*I   */
    float    *   correction                      /*  O */
);
M2(
    Ellipse      current_ellipse,                /*I   */
    int          tracker_number_in_test,         /*I/O */
    int          matrix_line_number,             /*I   */
    float        correction                      /*I   */
    float    *   quadratic_error,                /*  O */
    Ellipse  *   corrected_ellipse,              /*  O */
);
M1(
    Ellipse      current_ellipse,                /*I   */
    int          tracker_number_in_test,         /*I   */
    float        quadratic_error,                /*I   */
    Ellipse  *   final_current_ellipse,          /*  O */
    int      *   final_pattern_number            /*  O */
);
```

Algorithm 1: Signature of the application-specific sequential functions for the tracker application. completion time is higher in the multiprocessor case.

This negative speedup can be explained by the way the outer SCM skeleton is deployed on the network. In fact, with the runtime mechanism presented in Section 2.4, using two nodes does not provide more computing power but only creates communications. This is because one of the nodes is used as a dispatching process and is not performing useful computation at all. When the computation versus communication is small (as in the 170 points case), this effect can even be observed with 3 nodes because the time to communicate data to the two nodes doing computations destroys the potential gains of performing theses computations is parallel

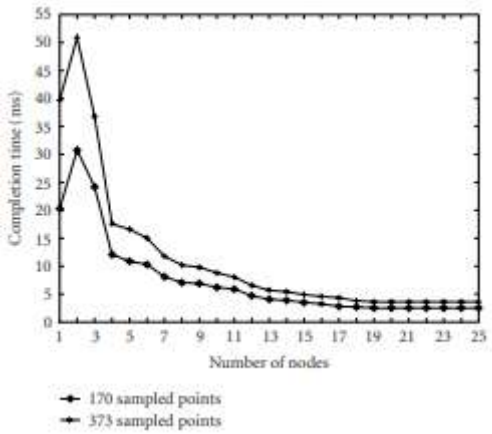

Figure 5 : Completion time for 170 and 373 sampled points for 3D face-tracking algorithm parallelisation on an Intel Celeron Beowulf machine.

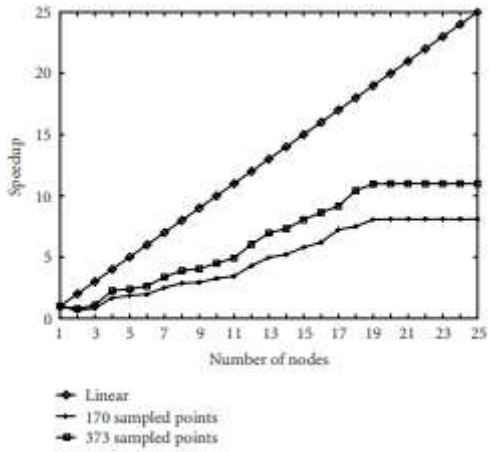

Figure 6: Speedup for 170 and 373 sampled points for 3D facetracking algorithm parallelisation on an Intel Celeron Beowulf machine.

Performance improves (albeit not linearly) as the number of nodes rises from 4 to 19. Parallel vector-matrix multiplications are carried out at this stage, which corresponds to the introduction of the inner SCM skeleton. There are initially 19 nodes in the final phase. Here, adding more computers makes little difference in performance. The reason for this is because each SCM skeleton has a predetermined data-parallel plan. When the number of accessible processing nodes is equal to this given parallelism degree, maximum efficiency is achieved.8 When there are more nodes available (19-32) than can be efficiently used via parallelism, efficiency drops. The kernel sequentializes certain processing's on some nodes (offering a type of "virtu alisation" method) when the number of nodes is modest (between 4 and 18). In a broader sense, the low efficiency results may be attributed to the parallel version's low computa tion versus communication ratio. The sequential implementation of the method was already very effective due to the small number of computationally demanding steps it required. Particularly at the inner parallelisation level, where the matrix multiplication is just a (p N) ma trix times the gray-level vector of size N, this holds true.

CONCLUSION

In this research, we provide a skeleton-based parallel programming environment that allows for nested skeletons, and we show how this new feature was used to parallelize a practical image processing application. One of the few known studies demonstrating the use of skeleton nesting facilities for the parallelization of a practical application, notably in the domain of image processing, is shown here. In fact, without this kind of skeleton combination, the 3D face-tracking method cannot be fully parallelized. This is because there is no way to combine the various layers of parallelisation into a single structure; instead, they must be managed via nested skeletons. However, the paper also demonstrates that Skipper-II's run-time mechanism incurs a large performance cost, particularly in situations when the ratio of computation to communication is low. In this situation, skeleton nesting is not an efficient transparent procedure.

REFERENCES

[1] D. Ginhac, J. Serot, and J.-P. D ´ erutin, "Fast prototyping of ´ image processing applications using functional skeletons on MIMD-DM architecture," in IAPR Workshop on Machine Vision Applications, pp. 468–471, Chiba, Japan, November 1998.

[2] D. Ginhac, Prototypage rapide d'applications parall`eles de vision artificielle par squelettes fonctionnels, Ph.D. thesis, Universite Blaise-Pascal, Clermont-Ferrand, France, January 1999. ´

[3] J. Serot, D. Ginhac, and J.-P. D ´ erutin, "Skipper: a skeleton- ´ based parallel programming environment for real-time image processing applications," in 5th International Conference on Parallel Computing Technologies (PACT '99), V. Malyshkin, Ed., vol. 1662 of Lecture Notes in Computer Science, pp. 296– 305, Springer-Verlag, Petersburg, Russia, September 1999.

[4] J. Serot, D. Ginhac, R. Chapuis, and J.-P. D ´ erutin, "Fast pro- ´ totyping of parallel-vision applications using functional skeletons," Machine Vision and Applications, vol. 12, no. 6, pp. 271– 290, 2001.

[5] R. Coudarcher, J. Serot, and J.-P. D ´ erutin, "Implementation ´ of a skeleton-based parallel programming environment supporting arbitrary nesting," in 6th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'01), F. Mueller, Ed., vol. 2026 of Lecture Notes in Computer Science, pp. 71–85, Springer-Verlag, San Francisco, Calif, USA, April 2001.

[6] M. Hamdan, G. Michaelson, and P. King, "A scheme for nesting algorithmic skeletons," in Proc. 10th International Workshop on Implementation of Functional Languages (IFL '98), C. Clack, T. Davie, and K. Hammond, Eds., pp. 195–212, London, UK, September 1998.

[7] G. Michaelson, N. Scaife, P. Bristow, and P. King, "Nested algorithmic skeletons from higher order functions," Parallel Algorithms and Applications, vol. 16, no. 2-3, pp. 181–206, 2001, Special issue on high level models and languages for parallel processing.

[8] M. Hamdan, A Combinational framework for parallel programming using algorithmic skeletons, Ph.D. thesis, HeriotWatt University, Department of Computing and Electrical Engineering, Edinburgh, UK, January 2000.

[9] F. Jurie and M. Dhome, "A simple and efficient template matching algorithm," in Proc. 8th IEEE International Conference on Computer Vision (ICCV '01), vol. 2, pp. 544–549, Vancouver, BC, Canada, July 2001.

[10] M. Cole, Algorithmic Skeletons: Structured Management of Parallel Computation, Pitman/MIT Press, London, UK, 1989.

[11] M. Cole, "Algorithmic skeletons," in Research Directions in Parallel Functional Programming, K. Hammond and G. Michaelson, Eds., pp. 289–303, Springer, UK, November 1999.

[12] J. Serot, "Embodying parallel functional skeletons: an exper- ´ imental implementation on top of MPI," in 3rd Intl Euro-Par Conference on Parallel Processing, C. Lengauer, M. Griebl, and S. Gorlatch, Eds., pp. 629–633, Springer, Passau, Germany, August 1999.

[13] N. Scaife, A dual source parallel architecture for computer vision, Ph.D. thesis, Heriot-Watt University, Department of Computing and Electrical Engineering, Edinburgh, UK, May 2000.

[14] T. Grandpierre, C. Lavarenne, and Y. Sorel, "Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors," in Proc. IEEE 7th International Workshop on Hardware/Software Codesign (CODES '99), pp. 74–79, Rome, Italy, May 1999.

[15] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A highperformance, portable implementation of the MPI message passing interface standard," Parallel Computing, vol. 22, no. 6, pp. 789–828, 1996.

[16] R. Coudarcher, Composition de squelettes algorithmiques: application au prototypage rapide d'applications de vision, Ph.D. thesis, LASMEA, Blaise-Pascal University, Clermont-Ferrand, France, December 2002.

[17] G. D. Hager and P. N. Belhumeur, "Efficient region tracking with parametric models of geometry and illumination," IEEE Trans. Pattern Anal. Machine Intell., vol. 20, no. 10, pp. 1025– 1039, 1998.

[18] F. Dellaert and R. Collins, "Fast image-based tracking by selective pixel integration," in International Conference on Computer Vision Workshop on Frame-Rate Vision, Corfu, Greece, September 1999.

[19] J. Serot, "Camlflow: a caml to data-flow translator," in ´ Trends in Functional Programming. Volume 2, S. Gilmore, Ed., pp. 129–141, Intellect Books, Bristol, UK, 2001.

[20] F. Marmoiton, F. Collange, P. Martinet, and J.-P. Derutin, "A ´ real time car tracker," in Proc. International Conference on Advances in Vehicle Control and Safety (AVCS '98), pp. 282–287, Amiens, France, July 1998.